# OOP

## SOLID + GRASP

JK

2020.04.01.

# Object-oriented Programming
# 객체 중심 프로그래밍

- 프로퍼티 property와 메소드 method
- 클래스 객체 class와 객체 인스턴스 instance
- 캡슐화 encapsulation
- 상속 inheritance
- 다형성 polymorphism
- 설계 패턴 design pattern

EBS 타큐멘터리 동과 서 1부. https://www.youtube.com/watch?v=J5hOkggR_nk
2부. https://www.youtube.com/watch?v=DInJUfD4hWk

# 아리스토텔레스

**클래스** **(객체) 인스턴스**

개념
일반화
무형
추상화
이론
구체
개별화
유형
구상화
실제

인터페이스
일반화, 추상화
구체화, 개별화
클래스
코드
일반화, 추상화
구체화, 개별화
객체
인스턴스

# 클래스와 객체 인스턴스

```
myPen = new NSPen()
```

속성 property + 행위 behavior

# 옛날 프로그램

in memory

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 3C | 3F | 78 | 6D | 6C | 20 | 76 | 65 | 72 | 73 | 69 | 6F | 6E | 3D | 22 | 31 |
| 16 | 2E | 30 | 22 | 20 | 65 | 6E | 63 | 6F | 64 | 69 | 6E | 67 | 3D | 22 | 55 | 54 |
| 32 | 46 | 2D | 38 | 22 | 20 | 73 | 74 | 61 | 6E | 64 | 61 | 6C | 6F | 6E | 65 | 3D |
| 48 | 22 | 6E | 6F | 22 | 3F | 3E | 0A | 3C | 64 | 6F | 63 | 75 | 6D | 65 | 6E | 74 |
| 64 | 20 | 74 | 79 | 70 | 65 | 3D | 22 | 63 | 6F | 6D | 2E | 61 | 70 | 70 | 6C | 65 |
| 80 | 2E | 49 | 6E | 74 | 65 | 72 | 66 | 61 | 63 | 65 | 42 | 75 | 69 | 6C | 64 | 65 |
| 96 | 72 | 33 | 2E | 43 | 6F | 63 | 6F | 61 | 54 | 6F | 75 | 63 | 68 | 2E | 53 | 74 |
| 112 | 6F | 72 | 79 | 62 | 6F | 61 | 72 | 64 | 2E | 58 | 49 | 42 | 22 | 20 | 76 | 65 |
| 128 | 72 | 73 | 69 | 6F | 6E | 3D | 22 | 33 | 2E | 30 | 22 | 20 | 74 | 6F | 6F | 6C |
| 144 | 73 | 56 | 65 | 72 | 73 | 69 | 6F | 6E | 3D | 22 | 31 | 33 | 31 | 32 | 32 | 2E |
| 160 | 31 | 36 | 22 | 20 | 74 | 61 | 72 | 67 | 65 | 74 | 52 | 75 | 6E | 74 | 69 | 6D |
| 176 | 65 | 3D | 22 | 69 | 4F | 53 | 2E | 43 | 6F | 63 | 6F | 61 | 54 | 6F | 75 | 63 |
| 192 | 68 | 22 | 20 | 70 | 72 | 6F | 70 | 65 | 72 | 74 | 79 | 41 | 63 | 63 | 65 | 73 |
| 208 | 73 | 43 | 6F | 6E | 74 | 72 | 6F | 6C | 3D | 22 | 6E | 6F | 6E | 65 | 22 | 20 |
| 224 | 75 | 73 | 65 | 41 | 75 | 74 | 6F | 6C | 61 | 79 | 6F | 75 | 74 | 3D | 22 | 59 |
| 240 | 45 | 53 | 22 | 20 | 6C | 61 | 75 | 6E | 63 | 68 | 53 | 63 | 72 | 65 | 65 | 6E |
| 256 | 3D | 22 | 59 | 45 | 53 | 22 | 20 | 75 | 73 | 65 | 54 | 72 | 61 | 69 | 74 | 43 |
| 272 | 6F | 6C | 6C | 65 | 63 | 74 | 69 | 6F | 6E | 73 | 3D | 22 | 59 | 45 | 53 | 22 |
| 288 | 20 | 75 | 73 | 65 | 53 | 61 | 66 | 65 | 41 | 72 | 65 | 61 | 73 | 3D | 22 | 59 |
| 304 | 45 | 53 | 22 | 20 | 63 | 6F | 6C | 6F | 72 | 4D | 61 | 74 | 63 | 68 | 65 | 64 |
| 320 | 3D | 22 | 59 | 45 | 53 | 22 | 20 | 69 | 6E | 69 | 74 | 69 | 61 | 6C | 56 | 69 |
| 336 | 65 | 77 | 43 | 6F | 6E | 74 | 72 | 6F | 6C | 6C | 65 | 72 | 3D | 22 | 30 | 31 |
| 352 | 4A | 2D | 6C | 70 | 2D | 6F | 56 | 4D | 22 | 3E | 0A | 20 | 20 | 20 | 20 | 3C |
| 368 | 64 | 65 | 70 | 65 | 6E | 64 | 65 | 6E | 63 | 69 | 65 | 73 | 3E | 0A | 20 | 20 |
| 384 | 20 | 20 | 20 | 20 | 20 | 20 | 3C | 70 | 6C | 75 | 67 | 49 | 6E | 20 | 69 | 64 |
| 400 | 65 | 6E | 74 | 69 | 66 | 69 | 65 | 72 | 3D | 22 | 63 | 6F | 6D | 2E | 61 | 70 |
| 416 | 70 | 6C | 65 | 2E | 49 | 6E | 74 | 65 | 72 | 66 | 61 | 63 | 65 | 42 | 75 | 69 |
| 432 | 6C | 64 | 65 | 72 | 2E | 49 | 42 | 43 | 6F | 63 | 6F | 61 | 54 | 6F | 75 | 63 |
| 448 | 68 | 50 | 6C | 75 | 67 | 69 | 6E | 22 | 20 | 76 | 65 | 72 | 73 | 69 | 6F | 6E |
| 464 | 3D | 22 | 31 | 33 | 31 | 30 | 34 | 2E | 31 | 32 | 22 | 2F | 3E | 0A | 20 | 20 |
| 480 | 20 | 20 | 20 | 20 | 20 | 20 | 3C | 63 | 61 | 70 | 61 | 62 | 69 | 6C | 69 | 74 |
| 496 | 79 | 20 | 6E | 61 | 6D | 65 | 3D | 22 | 53 | 61 | 66 | 65 | 20 | 61 | 72 | 65 |
| 512 | 61 | 20 | 6C | 61 | 79 | 6F | 75 | 74 | 20 | 67 | 75 | 69 | 64 | 65 | 73 | 22 |
| 528 | 20 | 6D | 69 | 6E | 54 | 6F | 6F | 6C | 73 | 56 | 65 | 72 | 73 | 69 | 6F | 6E |
| 544 | 3D | 22 | 39 | 2E | 30 | 22 | 2F | 3E | 0A | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 560 | 20 | 3C | 63 | 61 | 70 | 61 | 62 | 69 | 6C | 69 | 74 | 79 | 20 | 6E | 61 | 6D |
| 576 | 65 | 3D | 22 | 64 | 6F | 63 | 75 | 6D | 65 | 6E | 74 | 73 | 20 | 73 | 61 | 76 |
| 592 | 65 | 64 | 20 | 69 | 6E | 20 | 74 | 68 | 65 | 20 | 58 | 63 | 6F | 64 | 65 | 20 |
| 608 | 38 | 20 | 66 | 6F | 72 | 6D | 61 | 74 | 22 | 20 | 6D | 69 | 6E | 54 | 6F | 6F |
| 624 | 6C | 73 | 56 | 65 | 72 | 73 | 69 | 6F | 6E | 3D | 22 | 38 | 2E | 30 | 22 | 2F |
| 640 | 3E | 0A | 20 | 20 | 20 | 20 | 3C | 2F | 64 | 65 | 70 | 65 | 6E | 64 | 65 | 6E |
| 656 | 63 | 69 | 65 | 73 | 3E | 0A | 20 | 20 | 20 | 20 | 3C | 73 | 63 | 65 | 6E | 65 |
| 672 | 73 | 3E | 0A | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 3C | 21 | 2D | 2D | 56 |
| 688 | 69 | 65 | 77 | 20 | 43 | 6F | 6E | 74 | 72 | 6F | 6C | 6C | 65 | 72 | 2D | 2D |
| 704 | 3E | 0A | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 3C | 73 | 63 | 65 | 6E | 65 |
| 720 | 20 | 73 | 63 | 65 | 6E | 65 | 49 | 44 | 3D | 22 | 45 | 48 | 66 | 2D | 49 | 57 |
| 736 | 2D | 41 | 32 | 45 | 22 | 3E | 0A | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 752 | 20 | 20 | 20 | 3C | 6F | 62 | 6A | 65 | 63 | 74 | 73 | 3E | 0A | 20 | 20 | 20 |
| 768 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 3C | 76 | 69 |
| 784 | 65 | 77 | 43 | 6F | 6E | 74 | 72 | 6F | 6C | 6C | 65 | 72 | 20 | 69 | 64 | 3D |
| 800 | 22 | 30 | 31 | 4A | 2D | 6C | 70 | 2D | 6F | 56 | 4D | 22 | 20 | 73 | 63 | 65 |
| 816 | 6E | 65 | 4D | 65 | 6D | 62 | 65 | 72 | 49 | 44 | 3D | 22 | 76 | 69 | 65 | 77 |
| 832 | 43 | 6F | 6E | 74 | 72 | 6F | 6C | 6C | 65 | 72 | 22 | 3E | 0A | 20 | 20 | 20 |
| 848 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 864 | 20 | 3C | 76 | 69 | 65 | 77 | 20 | 6B | 65 | 79 | 3D | 22 | 76 | 69 | 65 | 77 |
| 880 | 22 | 20 | 63 | 6F | 6E | 74 | 65 | 6E | 74 | 4D | 6F | 64 | 65 | 3D | 22 | 73 |
| 896 | 63 | 61 | 6C | 65 | 54 | 6F | 46 | 69 | 6C | 6C | 22 | 20 | 69 | 64 | 3D | 22 |
| 912 | 5A | 65 | 35 | 2D | 36 | 62 | 2D | 32 | 74 | 33 | 22 | 3E | 0A | 20 | 20 | 20 |
| 928 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 944 | 20 | 20 | 20 | 20 | 20 | 3C | 72 | 65 | 63 | 74 | 20 | 6B | 65 | 79 | 3D | 22 |
| 960 | 66 | 72 | 61 | 6D | 65 | 22 | 20 | 78 | 3D | 22 | 30 | 2E | 30 | 22 | 20 | 79 |
| 976 | 3D | 22 | 30 | 2E | 30 | 22 | 20 | 77 | 69 | 64 | 74 | 68 | 3D | 22 | 33 | 37 |
| 992 | 35 | 22 | 20 | 68 | 65 | 69 | 67 | 68 | 74 | 3D | 22 | 36 | 36 | 37 | 22 | 2F |

한 덩어리에 data 와 procedure 섞여있음
계산 절차는 순차 실행만 가능했음

데이터와 실행 순서(절차)에 대한 구조화 필요성

# 추상화 메커니즘 #1

**프로시저 추상화 (procedure abstraction)**
- 무엇을 해야 하는지 절차에 대한 추상화

**기능 분해(Functional decomposition)**
**알고리즘 분해 (Algorithmic decomposition)**

**👉 절차 지향 - 구조적 프로그래밍 패러다임**

# 프로그래밍 패러다임 by 엉클밥

* 구조적 프로그래밍 : 제어 흐름의 직접적인 전환에 부과하는 규칙

  goto 구문 없이 반복문으로 제어 흐름에 대한 직접적인 전환

* 객체지향 프로그래밍 : 제어 흐름의 간적접인 전환에 부과하는 규칙

  다형성 +의존성 역전으로 제어 흐름에 대한 간접적인 전환 (모듈의 독립성)

* 함수형 프로그래밍 : 변수 할당에 부과되는 규칙

  불변성 / 가변성의 분리하는 변수의 독립성

  고차 함수로 동작에 대한 추상화

**이건 다른 수업에서...**

# GUI 프로그래밍과 기업용 응용 프로그램

https://en.wikipedia.org/wiki/PARC_(company)

# 추상화 메커니즘 #2

바바라 리스코프

데이터 추상화 (data abstraction)
- 무엇을 알아야 하는지 추상화

타입을 추상화(type abstraction) ➔ Abstract Data Type

- 타입 정의를 선언할 수 있어야 함
- 타입 인스턴스를 다루는 오퍼레이션 집합을 정의할 수 있어야 함
- 제공하는 오퍼레이션으로 조작하도록 보호할 수 있어야 함
- 타입에 대한 여러 인스턴스를 생성할 수 있어야 함

데이터 중심 프로시저 추상화 ➔ Object-oriented

- 상속과 다형성 지원

앨런 케이

# 추상 데이터 타입 vs 객체지향

Employee Type

| 오퍼레이션 | 정규 직원 | 아르바이트 직원 |
| --- | --- | --- |
| calculatePay() | basePay - (basePay * taxRate) | (basePay * timeCard) - (basePay * timeCard) * taxRage |
| monthlyBasePay() | basePay | 0 |

추상 데이터 타입

객체지향

# 객체 지향 타입으로 만들기

```
func calculatePay(taxRate: Double) -> Double {
    if (hourly) {
        return calculateHourlyPay(taxRate)
    }
    return calculateSalariedPay(taxRage)
}
```

# 다형성

Polymorphism

```swift
class Animal {
    func speak() {
        print("animal speak...")
    }
}

var animal = Animal()
animal.speak()

class Dog : Animal {
    override func speak() {
        print("dog - bow-wow")
    }
}

class Cat : Animal {
    override func speak() {
        print("cat - meow")
    }
}

var dog = Dog()
dog.speak()

var cat = Cat()
cat.speak()

var animalArray : [Animal] = [animal, dog, cat]
for x in animalArray {
    x.speak()
}
```

```swift
protocol AnimalProtocol {
    func speak()
}

struct Animal : AnimalProtocol {
    func speak() {
        print("animal speak")
    }
}

var animal = Animal()
animal.speak()

struct Dog : Animal {

}

struct Dog : AnimalProtocol {
    func speak() { print("dog - bow-wow") }
}

struct Cat : AnimalProtocol {
    func speak() { print("dog - bow-wow") }
}

var dog = Dog()
dog.speak()

var cat = Cat()
cat.speak()

var animalArray : [AnimalProtocol] = [animal, dog, cat]
for x in animalArray {
    x.speak()
}
```

# 클래스와 데이터 타입

추상 데이터 타입 vs 구체 데이터 타입 vs 인터페이스 타입

# 생물을 일반화 + 분류 classification

모든 생명체에 대해서 합의된 계통발생수(Consensus Phylogenetic Tree of All Life) 출처 : https://namu.wiki/w/계통분류학

출처 : http://www.falkhausen.de/Java-8/java.util/Collection-Hierarchy.html

여러분은
타입type vs 클래스class
선언하고 있나요?

```java
class Card {
  String suit;
  int rank;

  public Card() {
    suit = "diamond";
    rank = 1;
  }

  @Override
  public String toString() {
    return "Card{" +
      "suit='" + suit + '\'' +
      ", rank=" + rank +
      '}';
  }
}
```

타입/클래스를 포함, 활용, 소유, 생성

클래스에는 property 와 procedure 함께 선언

# 캡슐화 Encapsulation

정보 감추기 + 정보 접근 메서드로 추상화하기

복잡성

변경 가능성

```java
class Card {
  String suit;
  int rank;

  public Card(String suit, int rank) {
    this.suit = suit;
    this.rank = rank;
  }
}

class Player {
    List<Card> deck = new ArrayList();
    String name;

    public Player(String name) {
      this.name = name;
    }

    public Card lastCard() {
      return deck.get(deck.size()-1);
    }

    public void append(Card card) {
      deck.add(card);
    }

    public Card topRank() {
      Card maxRankCard = new Card("diamond", 1);
      for (Card card : deck) {
        if (card.rank > maxRankCard.rank) {
          maxRankCard = card;
        }
      }
      return maxRankCard;
    }
}
```

외부에서 접근 가능해서 안전하지 못한 속성 ❓

내부에서 필요한 의존적인 값을 외부에서 받아서 생성 ✅

내부에 배열이 있을 경우 추가, 접근, 탐색, 비교하는 코드가 늘어나서 복잡함 ❓

```java
class Card {
  private String suit;
  private int rank;

  public Card(String suit, int rank) {
    this.suit = suit;
    this.rank = rank;
  }

  //중간 생략
}

class Player {
    List<Card> deck = new ArrayList();
    String name;

    public Player(String name) {
      this.name = name;
    }

   //중간생략

    public Card topRank() {
      Card maxRankCard = new Card("diamond", 1);
      for (Card card : deck) {
        if (card.rank > maxRankCard.rank) {
          maxRankCard = card;
        }
      }
      return maxRankCard;
    }
}
```

private으로 바꾸면 어떻게 되나?

속성을 참조하던 부분에서 에러가 발생

```java
class Card {
  private String suit;
  private int rank;

  public Card(String suit, int rank) {
    this.suit = suit;
    this.rank = rank;
  }

  public int compare(Card other) {
    if (this.rank < other.rank)
      return -1;
    else if (this.rank == other.rank)
      return 0;
    else
      return 1;
  }
}
```

비교하는 메소드를 추가하고

```java
class Player {
    List<Card> deck = new ArrayList();
    String name;

    public Player(String name) {
      this.name = name;
    }
    //중간생략
    public Card topRank() {
      Card maxRankCard = new Card("diamond", 1);
      for (Card card : deck) {
        if (card.compare(maxRankCard) == 1) {
          maxRankCard = card;
        }
      }
      return maxRankCard;
    }
}
```

객체 단위로 (객체끼리) 비교하도록 개선

```java
class Player {
    List<Card> deck = new ArrayList();
    String name;

    public Player(String name) {
      this.name = name;
    }

    public Card lastCard() {
      return deck.get(deck.size()-1);
    }

    public void append(Card card) {
      deck.add(card);
    }

    public Card topRank() {
      Card maxRankCard = new Card("diamond", 1);
      for (Card card : deck) {
        if (card.compare(maxRankCard) == 1) {
          maxRankCard = card;
        }
      }
      return maxRankCard;
    }

    public Card lowRank() {
      Card minRankCard = new Card("diamond", 13);
      for (Card card : deck) {
        if (card.compare(minRankCard) == -1) {
          minRankCard = card;
        }
      }
      return minRankCard;
    }
}
```

가장 낮은 카드를 찾아야 하는 경우
또 탐색하는 메소드가 추가된다

```java
class CardDeck {
  private List<Card> deck = new ArrayList();

  public CardDeck() {
  }

  public Card last() {
    return deck.get(deck.size()-1);
  }

  public void append(Card card) {
    deck.add(card);
  }

  public Card findHighCard() {
    Card maxRankCard = new Card("diamond", 1);
    for (Card card : deck) {
      if (card.compare(maxRankCard) == 1) {
        maxRankCard = card;
      }
    }
    return maxRankCard;
  }

  public Card findLowCard() {
    Card minRankCard = new Card("diamond", 13);
    for (Card card : deck) {
      if (card.compare(minRankCard) == -1) {
        minRankCard = card;
      }
    }
    return minRankCard;
  }
}
```

```java
class Player {
    CardDeck deck = new CardDeck();
    String name;

    public Player(String name) {
      this.name = name;
    }

    public Card lastCard() {
      return deck.last();
    }

    public void append(Card card) {
      deck.append(card);
    }

    public Card topRank() {
      return deck.findHighCard();
    }

    public Card lowRank() {
      return deck.findLowCard();
    }
}
```

복잡성을 감추기

# 객체지향 설계

상위 객체와 하위 객체 추상화

기능 분해(Functional decomposition)과 복잡성을 감추기

(데이터+기능) 추상화

```
main()
input() -> Player() -> output()
CardDeck()
Card
```

(데이터+기능) 구체화

👉 **객체지향 프로그래밍 패러다임**

# 객체지향 설계와 구현 🤯

# 역할 Roles,
# 책임 Responsibility,
# 협력 Collaborations

## Class / Responsibility / Collaborator Cards

켄트 벡

EBS 타큐멘터리 동과 서

# 객체 속성을 가져오지 말고
# 객체가 일하도록 시켜라.

Don’t get it, Just ask them.

# 객체 설계 원칙 - SOLID

**1. SRP (Single-Responsibility Principle)**

소프트웨어 요소(클래스, 모듈, 함수 등)는 응집도 있는 하나의 책임을 갖는다. 클래스를 변경해야 하는 이유는 단지 **응집도**여야 한다.

**2. OCP (Open-Close Principle)**

소프트웨어 요소는 확장 가능하도록 열려있고, 변경에는 닫혀있어야 한다. 새 기능을 추가할 때 변경하지 말고 새 클래스나 함수를 만들어서 변하는 부분을 최소화한다.

**3. LSP (Liskov Substitution Principle)**

서브타입은 (상속받은) 기본 타입으로 대체가능해야 한다. 자식 클래스는 부모 클래스 동작(의미)를 바꾸지 않는다.

**4. DIP (Dependency-Inversion Principle)**

상위레벨 모듈은 하위레벨 모듈에 의존하면 안된다. (둘 다 추상화된 인터페이스에 의존해야 한다)

추상화는 구체화에 의존하면 안되고, 구체화는 추상화에 의존해야 한다.

**5. ISP (Interface-Segregation Principle)**

클라이언트 객체는 사용하지 않는 메소드에 의존하면 안된다.

# 응집도(cohesion)

LCOM4 = 2

LCOM4 = 1

응집도 지표 http://www.math.md/files/csjm/v25-n1/v25-n1-%28pp44-74%29.pdf

# 단일 책임 원칙 - 함수

# 단일 책임 원칙 - 객체

# 의존성 (dependency)

```java
class Player {
    CardDeck deck = new CardDeck();
    String name;

    public Player(String name) {
      this.name = name;
    }

    public Card lastCard() {
      return deck.last();
    }

    public void append(Card card) {
      deck.append(card);
    }

    public Card topRank() {
      return deck.findHighCard();
    }

    public Card lowRank() {
      return deck.findLowCard();
    }
}
```

구체 타입(concrete type)에 의존

# 열림-닫힘 원칙 - OCP

핵심 접근 방식은 간접참조 + 추상화

# 열림-닫힘 원칙 - OCP

```
private List<Card> deck = new ArrayList();
```

**Protocol-Oriented Programming**

# Code 설계

Piece
type
enum
PieceType
.pawn
.king
.queen
.rook
.bishop
.knight

# Value Object

```
enum Suit {
  Spade, Heart, Club, Diamond
}

class ScreenResolution {
  int width;
  int height;
}
```

DTO(Data Transfer Object)
Entity
VO(Value Object)
DS(Data Structure)
...
다양한 정의와 표현이 존재함

Data를 표현하고 Behavior를 갖고 있지 않는 경우

# 추상 타입 - OCP, LSP

Piece
type
PieceType
<Protocol>
Pawn
King
Queen
Rook
Bishop
Knight

# 인터페이스분리 원칙 - ISP

# 인터페이스분리 원칙 - ISP

# 인터페이스분리 원칙 - ISP

```
protocol GameManagerAction {
    func positionMap() -> Array<Array<Piece?>>
    func defaultFrame(path:(x : Int, y: Int)) -> CGRect
}
```

```
protocol GameManagerAction {
    func positionMap() -> Array<Array<Piece?>>
}
protocol GameManagerGeometric {
    func defaultFrame(for: (x : Int, y: Int)) -> CGRect
}

class ViewController: UIViewController {
    private var positionManager :
                  (GameManagerAction & GameManagerGeometric)!

    func setManager(_ manager :
                  GameManagerAction & GameManagerGeometric) {
        self.positionManager = manager
    }
```

# 의존성역전 원칙 - DIP

GameManager
<<interface>>
Player
제어흐름
(호출관계)
제어흐름
(호출관계)
<<interface>>
PlayerFactory
+makePlayer
Player
Impl
Player
Factory
+makePlayer

# 의존성역전 원칙 - DIP

```
interface CardDeckPlayable {
  public Card last();
  public void append(Card card);
  public Card findHighCard();
  public Card findLowCard();
}

class Player {
    CardDeckPlayable deck;
    String name;

    public Player(String name, CardDeckPlayable deck) {
      this.name = name;
      this.deck = deck;
    }
    //이하 생략
```

구체 타입 대신
추상 타입에 의존하고,

인스턴스 생성은
상위 모듈에서 소유

# 의존성역전 원칙 - DIP

**Constructor Injection**

```
class BackgroundView: UIView {
    private var presenter : BackgroundPresenter!
    convenience init(presenter: BackgroundPresenter) {
        self.init(frame: presenter.centralRect())
        self.presenter = presenter
    }
}
```

**Setter Injection**

```
protocol GameManagerAction {
    func positionMap() -> Array<Array<Piece?>>
    func defaultFrame(path:(x : Int, y: Int)) -> CGRect
}
class ViewController: UIViewController {
    private var manager : GameManagerAction!
    func setManager(_ manager : GameManagerAction) {
        self.positionManager = manager
    }
}
```

**Interface Injection**

```
protocol PieceImageSetter {
    func setImageBy(piece: Piece)
}
class PieceImageView: UIImageView, PieceImageSetter {

    func setImageBy(piece: Piece) {
        self.image = piece.type.image(color: piece.color.rawValue)
    }
}
```

# 의존성역전과 Composition Root

https://blog.ploeh.dk/2011/07/28/CompositionRoot/

https://vimeo.com/68378923

```
var t =
  new A1(
    new A2(
      new B1(
        new B2()),
      new A3()),
    new C1(
      new B3()));
```

# 책임 할당하기

- 데이터보다 행동을 먼저 결정하라

  외부에 제공하는 행동

- 협력이라는 문맥 안에서 책임을 결정하라

  클래스를 결정하고 그 클래스 책임을 찾기보다는
  메시지를 결정하고 이 메시지를 누구에게 전송할 지 찾아라.

- 책임 주도 설계와 GRASP 패턴

# 도메인 개념 : 카드게임

# GRASP 패턴

https://medium.com/@codesquad_yoda/스위프트와-grasp-패턴-d5e37a1bb5dc

1. 정보 담당자 Information Expert
2. 소유권한 Creator
3. 컨트롤러 Controller
4. 낮은 연결 Low Coupling
5. 높은 응집도 High Cohesion
6. 간접 참조 Indirection
7. 다형성 Polymorphism
8. 순수 조립 Pure Fabrication
9. 변화 보호 Protected Variations

# 도메인 개념 : 극장

# 정보 담당자 Information Expert

메시지를 전송할 객체는 무엇을 원하는가?

메시지를 수신할 적합한 객체는 누구인가?

# 정보 담당자 Information Export

# 높은 응집도와 낮은 결합도

# 변경과 응집도

높은 응집도(High Cohesion)

낮은 응집도(Low Cohesion)

# 변경과 결합도

높은 결합도(High Coupling)

낮은 결합도(Low Coupling)

# 창조자 Creator

- B가 A를 포함하거나 협력을 위해 조합하는 경우
- B가 A를 기록하는 경우
- B가 A를 긴밀하게(복잡하게) 사용하는 경우
- B가 A의 초기값을 갖고 있는 경우

# 구현을 통한 검증 Screening

메시지 응답을 위한 메소드 추가

```
class Screening {
    func reserve(with customer: Customer, audienceCount: Int) -> Reservation {
    }
}
```

---

책임에 필요한 인스턴스 변수 추가

```
class Screening {
    private var movie : Movie
    private var sequence : Int
    private var whenScreened : DateTime

    func reserve(with customer: Customer, audienceCount: Int) -> Reservation {
    }

}
```

---

책임에 필요한 내용을 구현

```
class Screening {
    private var movie : Movie
    private var sequence : Int
    private var whenScreened : DateTime

    func reserve(with customer: Customer, audienceCount: Int) -> Reservation {
        return Reservation(with: customer, screening: self, fee:calculateFee(audienceCount), audienceCount)
    }

    func calculateFee(int audienceCount) -> Money {
        return movie.calculateMovieFee(self).times(audienceCount)
    }
}
```

# 구현을 통한 검증 Movie

메시지 응답을 위한 메소드 추가

```
class Movie {
    func calculateMovieFee(screening: Screening) -> Money {
    }
}
```

책임에 필요한 인스턴스 변수 추가

```
class Movie {
    private let title : String
    private let runningTime : TimeInterval
    private let fee : Money
    private var discountConditions = Array<DiscountCondition>()

    private let movieType : MovieType
    private let discountAmount : Money
    private let discountPercent : Double

    func calculateMovieFee(screening: Screening) -> Money {
    }
}
```

# 구현을 통한 검증 Movie

책임에 필요한
내용을 구현

```
class Movie {
    enum MovieType {
        case AmountDiscount
        case PercentDiscount
        case NoneDiscount
    }

    private let title : String
    private let runningTime : TimeInterval
    private let fee : Money
    private var discountConditions = Array<DiscountCondition>()

    private let movieType : MovieType
    private let discountAmount : Money
    private let discountPercent : Double

    func calculateMovieFee(screening: Screening) -> Money {
        if isDiscountable(for: screening) {
            return fee.minus(calculateDiscountAmount())
        }
        return fee
    }

    private func isDiscountable(for: Screening) -> Bool {
        return discountConditions.filter{ $0.isSatisfied(by: screening) }.count > 0
    }
}
```

# 구현을 통한 검증 DiscountCondition

메시지 응답을 위한 메소드 추가

```
class DiscountCondition {
    func isSatisfied(by screening: Screening) -> Bool {
    }
}
```

---

책임에 필요한 내용을 구현

```
class DiscountCondition {
    private let type : DiscountConditionType
    private let sequence : Int
    private let dayOfWeek : DayOfWeek
    private let startTime : Date
    private let endTime : Date

    func isSatisfied(by screening: Screening) -> Bool {
        if type == .period {
            return isSatisfiedByPeriod(screening)
        }
        return isSatisfiedBySequence(screening)
    }

    private func isSatisfiedByPeriod(_ screening: Screening) -> Bool {
        return dayOfWeek.equals(screening.whenScreened.dayOfWeek) &&
            startTime.compare(to: screening.whenScreended.toLocalTime() <= 0) &&
            endTime.isAfter(to: screening.whenScreended.toLocalTime() >= 0)
    }

    private func isSatisfiedBySequence(_ screening: Screening) -> Bool {
        return sequence == screening.sequence
    }
}
```

# 구현을 통한 검증 Screening & DiscountCondition

정보 제공을 위한 접근성 변경 →

```
class Screening {
    private var movie : Movie
    private(set) var sequence : Int
    private(set) var whenScreened : DateTime

    func reserve(with customer: Customer, audienceCount: Int) -> Reservation {
        return Reservation(with: customer, screening: self, fee:calculateFee(audienceCount), audienceCount)
    }

    func calculateFee(int audienceCount) -> Money {
        return movie.calculateMovieFee(self).times(audienceCount)
    }
}
```

---

책임에 필요한 타입 추가 →

```
class DiscountCondition {
    enum DiscountConditionType {
        case Sequence, Period
    }
    private let type : DiscountConditionType
    private let sequence : Int
    private let dayOfWeek : DayOfWeek
    private let startTime : DateTime
    private let endTime : DateTime

    //… 생략
```

# SRP (단일 책임 원칙)

- 새로운 할인 조건 추가
- 순번 조건을 판단하는 로직 변경
- 기간 조건을 판단하는 로직이 변경되는 경우

**응집도 판단하기**

✔클래스가 하나 이상의 이유로 변경해야 한다면 응집도 낮은 것이다.
변경의 이유를 기준으로 클래스를 분리하라.

✔클래스의 인스턴스를 초기화하는 시점이 아니라, 나중에 서로 다른 속성을 초기화하고 있다면 응집도가 낮은 것이다.
초기화하는 속성 그룹을 기준으로 클래스를 분리하라.

✔메소드 그룹이 속성 그룹을 사용하는지 여부로 나뉜다면 응집도가 늦은 것이다.
속성 그룹을 기준으로 클래스를 분리하라.

# 타입 분리하기 PeriodCondition - SequenceCondition

```
class PeriodCondition {
    private let dayOfWeek : DayOfWeek
    private let startTime : Date
    private let endTime : Date

    init(dayOfWeek: DayOfWeek, startTime: DateTime, endTime: DateTime) {
        self.dayOfWeek = dayOfWeek
        self.startTime = startTime
        self.endTime = endTime
    }

    func isSatisfied(by screening: Screening) -> Bool {
        return dayOfWeek.equals(screening.whenScreened.dayOfWeek) &&
            startTime.compare(to: screening.whenScreended.toLocalTime() <= 0) &&
            endTime.isAfter(to: screening.whenScreended.toLocalTime() >= 0)
    }
}
```

---

```
class SequenceCondition {
    private let sequence : Int

    init(with sequence: Int) {
        self.sequence = sequence
    }

    func isSatisfied(by screening: Screening) -> Bool {
        return sequence == screening.sequence
    }
}
```

# 두 종류 객체와 협력하는 Moive

# 타입 분리하기 Movie

```
class Movie {
    enum MovieType {
        case AmountDiscount
        case PercentDiscount
        case NoneDiscount
    }
    private let title : String
    private let runningTime : TimeInterval
    private let fee : Money
    private let movieType : MovieType
    private let discountAmount : Money
    private let discountPercent : Double

    private var periodConditions = Array<PeriodCondition>()
    private var sequenceConditions = Array<SequenceCondition>()

    func calculateMovieFee(screening: Screening) -> Money {
        if isDiscountable(for: screening) {
            return fee.minus(calculateDiscountAmount())
        }
        return fee
    }

    private func mayPeriodConditions(with screening: Screening) -> Bool {
        return periodConditions.filter{ $0.isSatisfied(by: screening) }.count > 0
    }

    private func maySequenceConditions(with screening: Screening) -> Bool {
        return sequenceConditions.filter{ $0.isSatisfied(by: screening) }.count > 0
    }

    private func isDiscountable(for: Screening) -> Bool {
        return mayPeriodConditions(with: screening) || maySequenceConditions(with: screening)
    }
```

# 다형성 적용하기

DiscountCondition

```
protocol DiscountCondition {
    func isSatisfied(by screening: Screening) -> Bool
}

class PeriodCondition : DiscountCondition {
}

class SequenceCondition : DiscountCondition {
}
```

# 다형성 적용하기 Movie

```
class Movie {
    enum MovieType {
        case AmountDiscount
        case PercentDiscount
        case NoneDiscount
    }
    private let title : String
    private let runningTime : TimeInterval
    private let fee : Money
    private let movieType : MovieType
    private let discountAmount : Money
    private let discountPercent : Double

    private var discountConditions = Array<DiscountCondition>()

    func calculateMovieFee(screening: Screening) -> Money {
        if isDiscountable(for: screening) {
            return fee.minus(calculateDiscountAmount())
        }
        return fee
    }

    private func isDiscountable(for: Screening) -> Bool {
        return discountConditions.filter{ $0.isSatisfied(by: screening) }.count > 0
    }
```

# 변경으로부터 보호 Protected Variations

# 다형성 적용하기 Movie

# 다형성 가치

self
1. 메시지 전송
[message]
operation
anObject
method
2. 오퍼레이션
호출
3. 메소드
실행

# 그외 규칙들

다양한 방법

# 디미터 법칙

* 객체들 협력 경로를 제한하면 결합도를 낮출 수 있다

➔ 낯선 자에게 말하지 말라. 오직 인접한 이웃하고만 말하라

* 아래 조건을 만족하는 인스턴스에만 메시지를 전송하자
  - self 객체
  - 메소드의 매개변수
  - self 내부 속성
  - self 내부 속성으로 선언한 콜랙션 요소
  - 메소드 내에서 생성한 지역 객체

# 디미터 법칙 예시 before

```
class ReservationAgency {
    func reserve(screening : Screening, customer: Customer, audienceCount: Int) -> Reservation {
        let movie = screening.getMovie()
        var discountable = false
        for condition in movie.getDiscountConditions() {
            if condition.getType() == .Period {
                discountable = screening.whenScreened().getDayOfWeek().equals(codition.getDayOfWeek()) &&
                    condition.getStartTime().compareTo(screening.whenScreend().toLocalTime()) <= 0 &&
                    condition.getEndTime().compareTo(screening.whenScreend().toLocalTime()) <= 0 &&
            }
            else {
                discountable = condition.getSequence() == screening.getSequence()
            }

            if discountable { break }
        }

        var fee : Money
        if discountable {
            var discountAmount = Money.ZERO
            switch movie.getMovieType() {
            case AMOUNT_DISCOUNT:
                discountAmount = movie.getDiscountAmount()
            case PERCENT_DISCOUNT:
                discountAmount = movie.getFee().times(movie.getDiscountPercent())
            case NONE_DISCOUNT:
                break
            }
            fee = movie.getFee().minus(discountAmount).times(audienceCount)
        }
```

# 디미터 법칙 예시 after

```
class ReservationAgency {
    func reserve(screening : Screening, customer: Customer, audienceCount: Int) -> Reservation {
        let fee = screening.calculateFee(audienceCount)
        return Reservation(custom, screening, fee, audienceCount)
    }
}
```

# 디미터 법칙 함정 exception

```
class PeriodCondition : DiscountCondition {
    private let dayOfWeek : DayOfWeek
    private let startTime : Date
    private let endTime : Date

    init(dayOfWeek: DayOfWeek, startTime: DateTime, endTime: DateTime) {
        self.dayOfWeek = dayOfWeek
        self.startTime = startTime
        self.endTime = endTime
    }

    func isSatisfied(by screening: Screening) -> Bool {
        return dayOfWeek.equals(screening.whenScreened.dayOfWeek) &&
            startTime.compare(to: screening.whenScreended.toLocalTime() <= 0) &&
            endTime.isAfter(to: screening.whenScreended.toLocalTime() >= 0)
    }
}
```

```
class PeriodCondition : DiscountCondition {
    private let dayOfWeek : DayOfWeek
    private let startTime : Date
    private let endTime : Date

    // 중간 생략
    func isSatisfied(by screening: Screening) -> Bool {
        return screening.isDiscountable(dayOfWeek: dayOfWeek, startTime: startTime, endTime: endTime)
    }
}

extension Screening {
    func isDiscountable(dayOfWeek: DayOfWeek, startTime: DateTime, endTime: DateTime) -> Bool {
        return dayOfWeek.equals(whenScreened.dayOfWeek) &&
            startTime.compare(to: whenScreended.toLocalTime() <= 0) &&
            endTime.isAfter(to: whenScreended.toLocalTime() >= 0)
    }
}
```

# 명령-쿼리 분리 원칙

* 프로시저와 함수

➔ 프로시저 : 부수효과를 발생할 수 있지만 값을 반환할 수 없다

* 객체 상태를 변경하는 명령은 반환값을 가질 수 없다 (mutable)

➔ 함수 : 값을 반환할 수 있지만 부수효과를 발생시킬 수 없다

* 객체 정보를 반환하는 쿼리는 상태를 변경할 수 없다 (immutable)

참조 투명성

So: both OOP and functional computation can be completely compatible (and should be!). There is no reason to munge state in objects, and there is no reason to invent “monads” in FP. We just have to realize that “computers are simulators” and figure out what to simulate.

- Alan Kay